Report on the 1 st International Workshop on Debugging in Model-Driven Engineering (MDEbug'17)

Simon van Mierlo, Erwan Bousse, Hans Vangheluwe, Manuel Wimmer, Clark Verbrugge, Martin Gogolla, Matthias Tichy, Arnaud Blouin

Report on the 1st International Workshop on Debugging in Model-Driven Engineering (MDEbug'17)

Simon Van Mierlo*, Erwan Bousse†, Hans Vangheluwe*‡§, Manuel Wimmer†, Clark Verbrugge‡, Martin Gogolla¶, Matthias Tichy‖ and Arnaud Blouin††
* University of Antwerp, Belgium
Email: simon.vanmierlo@uantwerpen.be
Email: hans.vangheluwe@uantwerp.be
† TU Wien, Austria
Email: erwan.bousse@tuwien.ac.at
Email: wimmer@big.tuwien.ac.at
‡ McGill University, Canada
Email: clump@cs.mcgill.ca
§ Flanders Make vzw, Belgium
¶ University of Bremen, Germany
Email: gogolla@informatik.uni-bremen.de
‖ University of Ulm, Germany
Email: matthias.tichy@uni-ulm.de
†† INSA Rennes, France
Email: arnaud.blouin@irisa.fr

***Abstract*—System developers spend a significant part of their time *debugging* systems (*i.e.,* locating and fixing the cause of failures observed through *verification and validation* (V&V)). While V&V techniques are commonly used in model-driven engineering, locating and fixing the cause of a failure in a modelled system is most often still a manual task without tool-support. Although debugging techniques are well-established for programming languages, only a few debugging techniques and tools for models have been proposed. Debugging models faces various challenges: handling a wide variety of models and modelling languages; adapting debugging techniques initially proposed for programming languages; tailoring debugging approaches for the domain expert using the abstractions of the considered language. The aim of the first edition of the MDEbug workshop was to bring together researchers wanting to contribute to the emerging field of debugging in model-driven engineering by discussing new ideas and compiling a research agenda. This paper summarizes the workshop's discussion session and distils a list of challenges that should be addressed in future research.**

I. Introduction

Model-Driven Engineering (MDE) aims at coping with the complexity of systems by separating concerns through the use of models, each representing a particular aspect of a system. In the past years, significant effort has been directed towards providing early verification and validation (V&V) techniques to determine whether or not a set of models fulfils a set of properties (*e.g.,* [1], [2], [3], [4]). By identifying the properties that are not satisfied, such techniques are able to discover and observe the *failures* (or *bugs*) of a system. Yet, once a failure has been observed, it is then necessary to identify why the failure occurs (*i.e.*, the defect that caused the failure), and how to modify the models to remove the cause of the failure. These two tasks constitute the core of the *debugging* activity [5], [6].

To illustrate this activity, Figure 1 presents an overview of a typical system design process. First, a set of properties that the system has to satisfy is defined. Then, a system is designed as a collection of models that must satisfy these properties. To check that the properties are satisfied by the models, a wide range of verification and validation (V&V) techniques are available, such as theorem proving, symbolic execution, model checking, real-time simulation, and testing. Depending on the approach, it might be found that a property is satisfied ("pass"), not satisfied[1] ("failure"), or that the result is inconclusive ("unknown")—for example when the chosen technique cannot prove the property in a reasonable time frame. At that point, however, the *cause* of the failure (also called the *defect*) must still be identified (*i.e.,* which parts of the models are causing the observed failure). A failure may also be observed if the properties were wrongly specified, in which case the properties themselves have to be fixed. Once they are identified, the cause of the failures must be fixed by changing either the models or the properties. Locating and fixing the cause of a failure can be accomplished manually given a good understanding

[1]Note that it is also possible to consider *potential* failures when the considered V&V technique may give false positives (*e.g.,* static code analysis)

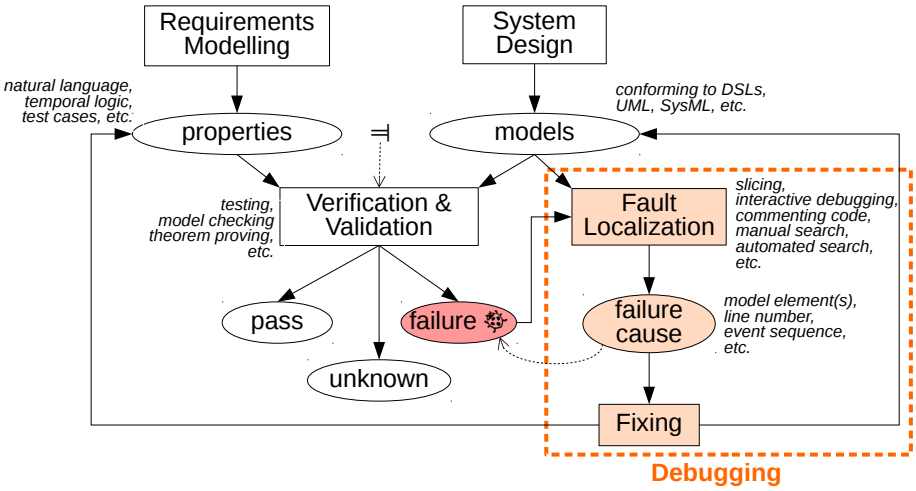

Figure 1. The debugging activity in the system design process.

of the models. A wide range of *debugging techniques* can be used to assist developers in finding the cause of the problem. For example, *interactive debugging* techniques [7], [8], [9] can be used to observe and control the execution of behavioural models in an interactive fashion (*e.g.,* using breakpoints, stepping operators, or by inspecting properties) Other techniques may provide (semi-)automated fault localization, for instance using symbolic execution [10], or model slicing [11]. More recently, omniscient debugging techniques [12], [13] allow one to explore execution states both backwards and forwards, and live modelling [14] allows one to change the model and immediately observe the effect.

II. Workshop Context and Goals

While debugging techniques and tools are very common for programming languages, very few debugging tools and techniques are available when it comes to models. Hence, modellers often have to resort to ad-hoc methods, such as inspecting and debugging the code generated from models. Although this allows the reuse of established and well-researched program debugging techniques, it is not ideal since the developer has to switch contexts and must understand both the mapping to and the semantics of the underlying implementation language. Dedicated debugging support for modelling languages has potential to reduce or eliminate the need for this kind of context switching, and is an essential part of allowing a developer to remain in the modelling paradigm throughout the full development process.

In this context and scope, the goals of the MDEbug workshop were to:

- bring together interested researchers to optimize the research effort and establish collaborations;
- provide a forum for researchers to share new experiences, ideas and early results on the topic of debugging in model-driven engineering;
- define the scope of debugging within model-driven engineering;
- identify gaps in the current body of research.

III. Program

The full-day workshop took place on September 17, 2017 as part of the satellite events of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017) conference in Austin, Texas. The workshop started with a keynote presented by Andrei Chiș from *feenk* on the topic of Moldable Debugging. We received six submissions, and five submissions were accepted after a review process in which each paper was reviewed by at least three members of the program committee. One of the accepted submission was a full research paper, while two were tool demonstration papers, and two were position papers—including one by the keynote speaker that summarized his keynote presentation. The complete list of papers can be found below, in the order of their presentation at the workshop:

1) "Moldable Debugging (Position paper)" by Andrei Chiș and Tudor Girba
2) "Transformations Debugging Transformations" by Maris Jukss, Clark Verbrugge and Hans Vangheluwe
3) "Towards Debugging the Matching of Henshin Model Transformations Rules (Position paper)" by Matthias Tichy, Luis Beaucamp and Stefan Kögel
4) "Domain-Level Debugging for Compiled DSLs with the GEMOC Studio (Tool demonstration)" by Erwan Bousse, Tanja Mayerhofer and Manuel Wimmer
5) "Debugging Non-Determinism: a Petrinets Modelling, Analysis, and Debugging Tool (Tool demonstration)" by Simon Van Mierlo and Hans Vangheluwe

The afternoon session of the workshop was reserved for an interactive discussion on the wide topic of debugging in model-driven engineering. We summarize these discussions in Section V.

All information can be found on the workshop website: https://msdl.uantwerpen.be/conferences/MDEbug. This includes the slides of all presentations given at the workshop.

IV. Program Committee

The program committee of MDEbug 2017 was composed of 27 researchers and experts in the domains of modeling, debugging, and model execution, coming from 12 different countries. We sincerely thank the program committee members and external reviewers for their time in reviewing and discussing the submitted papers.

- Mauricio Alférez, Siemens, Belgium
- Shaukat Ali, Simula Research Laboratory, Norway
- Vasco Amaral, Universidade Nova de Lisboa, Portugal
- Reda Bendraou, Université Paris Ouest Nanterre la Défense, France
- Francis Bordeleau, CMind, Canada
- Benoit Combemale, IRISA, Université de Rennes 1, France
- Jonathan Corley, University of West Georgia, USA
- Andrea d'Ambrogio, University of Rome Tor Vergata, Italy
- Julien Deantoni, University of Nice-Sophia Antipolis, France
- Davide Di Ruscio, University of L'Aquila, Italy
- Holger Giese, Hasso-Plattner-Institut, Germany
- Martin Gogolla, University of Bremen, Germany
- Jeff Gray, University of Alabama, USA
- Robert Heinrich, Karlsruher Institute of Technology, Germany
- Sebastian Herzig, Caltech/Jet Propulsion Laboratory, USA
- Levi Lucio, Fortiss, Germany
- Tanja Mayerhofer, TU Wien, Austria
- Tim Molderez, Vrije Universiteit Brussel, Belgium
- Patrizio Pelliccione, Chalmers University of Technology and University of Gothenburg, Sweden
- Arend Rensink, Universiteit Twente, The Netherlands
- Bran Selic, Malina Software Corporation, Canada
- Eugene Syriani, University of Montreal, Canada
- Jérémie Tatibouët, CEA, France
- Massimo Tisi, Institut Mines-Telecom Atlantique, France
- Javier Troya, University of Seville, Spain
- Antonio Vallecillo, Universidad de Málaga, Spain
- Tijs van der Storm, Centrum Wiskunde & Informatica (CWI), The Netherlands

V. Summary of the Discussions

This section summarizes the content of the discussions that took place during the workshop, in the form of a list of topics. Each topic covers an aspect of debugging in model-driven engineering that has open research challenges. Therefore, these challenges can be used as a basis for future research, or for organizing discussions during a future workshop.

A. Defining the state of the art in software debugging

Debugging is not limited to MDE. Debugging embraces various activities, techniques, and application domains in software engineering prior to MDE, such as debugging object-oriented programs. Yet, to the best of our knowledge no recent state of the art has been published on debugging in software engineering. The interested reader has to explore the literature of various techniques applied in a debugging context, such a program slicing.

A direction of research can focus on designing a comprehensive survey on debugging in software engineering. The outcome(s) of this work would permit to clearly identify the challenges shared or specific to MDE debugging.

B. Clarifying debugging terminology, classifying approaches

Due to a rather rich history in the field of computer science, the words *bug*, *debugging*, and *debugger* have been used in a large number of contexts with diverse meanings. For instance, some might call "bug" the wrong behaviour of a program, while some others consider that the "bug" is the piece of code responsible for the misbehaviour. Likewise, while "debugging" literally means "removing a bug", in some contexts it means "finding the bug", while in some others it means "executing the model/program in an interactive debugger". Lastly, the word "debugger" is used on a daily basis to designate an interactive debugger (*e.g.,* `gdb`, or the Eclipse Java debugger), while more generally it can mean "a tool that can be used for debugging". Overall, to avoid confusion, it appears that the community must either agree on a precise terminology, or at least acknowledge that these words may convey different meanings depending on the context.

In addition to vocabulary issues, there is some confusion on the different kinds of debugging approaches that exist, and what qualifies as debugging, understanding, or neither. For instance, using an interactive debugger to explore the different states of a behavioural model step by step can be useful both for debugging (*i.e.,* finding and fixing a bug), or simply to better understand the model. Likewise, a model slicer can be used to debug a model but is commonly not called a debugger, nor is model slicing classified as a debugging technique since it can be used for other purposes as well. It would be of great benefit for the community to reach a classification of possible debugging techniques and to understand the similarities or differences between them (*e.g.,* interactive or not, automated or manual, static or dynamic, language-specific or generic). The consensus at the workshop seemed to be that any technique used during the "debugging phase" (denoted by the dashed orange line in Figure 1) can be called a debugging technique. Among many others, this includes print statements, interactive debugging, and model slicing. It is not so much the technique, but rather the *intent* of the use of the technique that defines it as a debugging technique.

C. Abstraction gap and translational semantics

Semantics for modelling languages can be defined in a variety of ways [15]. One approach is to develop a trans-

formation that maps models of the language onto another language with known semantics. A popular example is a code generator that generates program code implementing the model's semantics. Models can be debugged naively using an interactive debugger for the programming language to debug the generated code or the model interpreter in case of an execution by interpretation. The "semantic gap" between the source and target language, however, obscures the abstractions of the source language; the debugging is not performed at the same abstraction level as the development. Certain approaches reuse the target language's interactive debugger and translate debugging operations on the source language forward onto debugging operations of the target language, and results of the target language interactive debugger backward onto results of the source language (*e.g.,* [16]). The semantic gap, however, makes it difficult to translate debugging results back onto abstractions of the source language.

In a general sense, the mapping between source and target languages is not always one-to-one. One open problem is how to handle such situations. Should the target language concepts be available to the modeller? Should additional domain concepts be defined? Or should one consider that there is a bug in the translational semantics themselves?

D. Impact of the observation on the execution

While some approaches reuse a target language interactive debugger to provide interactive debugging for modelling languages with translational semantics, instrumentation offers another way of debugging models. For instance, it is possible to instrument the model with specific "traps" that interrupt the normal flow of execution to provide source language interactive debugging operations [17]. In the case of operational semantics, interactive debugging or generic control operations can also directly be added to the interpreter [8], [18], [9]. In both cases, however, the instrumentation might change the semantics in a way that interferes with the normal execution semantics of the model. This means bugs can appear and/or disappear after instrumentation, or due to the instrumentation itself. In this case, bug-preserving instrumentation methods appear to be necessary [19].

E. Debugging forward or backward

Interactive debugging is usually performed forwards: from an initial state, one explores the execution trace according to the semantics of the considered modelling language. It is possible to extend this approach to allow the developer to step *backwards* in an execution trace, leading to so-called "omniscient debugging" [12], [13]. Being able to interactively explore execution states both forward and backward, hence allowing to freely explore the state space, yields many benefits regarding usability, as it may prevent from having to restart the complete execution to re-visit some suspect state.

To extend even further the possibilities of omniscient debugging, an interesting research direction is to study the potential for starting from a faulty model state to then compute the possible past execution paths that can lead up to that state, thereby debugging entirely backwards. This is related to program or model slicing, where the debugger produces a set of statements that can affect the values of a run-time state.

F. Debugging in the presence of black-box components

Debuggers may assume that the model source (a text document, diagram, etc.) is available. This might not always be true. In certain cases the source of the model might be hidden and provided as a compiled "black-box". If so, the control an interactive debugger can exert over such a black box is limited: in program interactive debugging, these components are often ignored (or "stepped over") and assumed to be correct. Failures that only appear when composing or integrating black-box components can be difficult to debug: the failure might actually be caused by an interaction in one of the components that is impossible to detect without observation of the state and control of the execution. If it is impossible to provide the full source (*e.g.,* in case of intellectual property protection), a grey-box approach might provide an interface to the component that allows a debugger to interact with it. This requires finding a balance between exposing enough detail and protecting the full details of the model.

G. Usability and fitness to the domain

Debuggers must be developed for developers and modellers, which are often domain experts with no experience with the technologies used to implement the modelling environments that they use. The usability of a debugger is therefore a key issue and may cover many aspects. In the case of an interactive debugger, such aspects may include the following:

- The representation of the current execution state has to fit the mental model of the developer. For debugging visual languages, this representation can be close to the visual abstractions provided by the language. In the presence of additional run-time information, however, additional abstractions may have to be provided.
- Similarly, not only the debugging state but also definitions of breakpoints need to align with the language. Here, domain-specific condition languages on states and state sequences need to be provided which enable the domain expert to easily specify conditions on the state according to the domain-specific needs.
- The debugging operations provided have to fit the needs of the developers. They have to fit the developer's mental model of the language's semantics: a logical "step" does not necessarily correspond to a debugger or simulator "step".
- Suitable representations for the execution traces and their states have to be found. Especially for concurrent, non-deterministic formalisms and/or acausal languages, these might differ significantly from the usual sequential representation.

Furthermore, while the usual concepts of interactive debugging (state, steps, breakpoints, etc.) are obvious to software engineers and can be easily used also for models (*e.g.,* [16], [20], [12]), domain experts might not be well-versed using

interactive debuggers. For example, a debugger for a modelling language might provide interactive tutorials and similar features to enable domain experts to quickly learn how to debug the model. In essence, this usability aspect needs to be evaluated with respect to the needs of the domain experts.

While there is already existing and encouraging work on domain-specific debugging [21], [22], usability is a key challenge for all debugging approaches.

H. Reuse of debuggers among languages

Many formalisms have common aspects they share: syntax, semantics, visualization. In language engineering, efforts are made to reuse different parts of language definitions [23], [24]. Likewise, efforts can be made to share tools among different languages, such as debuggers [25], [12]. For instance, interactive debuggers often share similar features and concepts, such as steps, breakpoints, state inspection, or state manipulation. One research direction is therefore to investigate how different parts and the logic of debuggers can be reused and shared to avoid the effort of implementing a new debugger from scratch for each new language. General requirements for a common debugger interface are also interesting for example in situations when different debuggers for one modelling language are available and a project wants to change the debugger.

I. Interactive debugging for declarative languages

A declarative language generally provides concepts that do not explicitly show or define in which order a conforming model will be executed. A prime example of this is a declarative model transformation: it defines the relation between input elements and output elements, but does not define how the transformation is executed. The engine responsible for executing the transformation is defined operationally, however. To debug such declarative languages interactively, one can ask whether we have to expose those operational semantics to the developer (in the form of steps, for example), and if so, to what extent. Since the language is declarative, the developer does not necessarily know its operational semantics, and exposing them might require a mental leap. Alternatively, debugging operations that are far away from the operational semantics but closer to the mental model of the developer might be more appropriate (*e.g.,* by showing all maximal parts of a declarative model transformation which are still applicable to understand why the full declarative model transformation is not applicable). This would lead to debugging functionality that considers the constituents of the declarative model (*e.g.,* debugging the formulas being part of the declarative model).

J. Debugging hardware systems

While many systems are deployed as software, synthesis to hardware is also possible. When a failure occurs in a synthesized hardware part, it can be necessary to debug this part to identify the cause of the failure, using physical probes to read the state and control of the system. Many tools and techniques exist for hardware monitoring, information, and analysis. While we currently often transpose software debugging concepts, a valuable line of research can look at hardware debugging techniques and transpose them to models.

K. Debugging structural models

Debugging is often defined as finding the cause of some observed faulty *behaviour* of a system, and changing the system to avoid this behaviour in the future. This suggests that debugging only makes sense for behavioural models, and therefore for modelling languages with execution semantics implemented by a code generator or an interpreter.

Yet, a wide range of models focus only on *structural* aspects of systems, and such models may also contain errors. For example, a metamodel can be considered faulty and too restrictive if a supposed valid model does not conform to this metamodel. Another example is a structural model of a building that does not conform to a regulation. "Debugging" such models would consist in searching for static constraints that are not well defined. This might require query and expression support to ask the conformance checker why or why not certain constraints were violated. Analysis for descriptive elements like constraints in terms of formulas must be provided (*e.g.,* pointing to the failing subformula when a conjunction fails). A possible research direction would be to study or define debugging approaches fully dedicated to structural models.

L. Debugging in the context of faulty semantics

While debugging commonly focuses on finding problems inside models, another possible cause for failures are problems in the *semantics* of the considered language (*e.g.,* errors in the interpreter or in the code generator). Faulty semantics can lead to invalid or inconsistent debugging results. This can have different causes, such as:

- The semantics is correctly specified, but wrongly implemented, which means that the semantics itself must be debugged.
- The semantics is incorrectly specified, which means that the set of properties that specify the language must be clarified.
- The semantics is both correctly specified and implemented, but is not well understood by the modeller. This is a subtler problem, and one where debugging techniques can be useful for "model understanding", or in this case "semantics understanding", possibly by exposing certain details of the semantics in the debugging process.

M. Identifying "innocent" model elements

Debugging aims at searching the cause of some observed failures within models or properties. In other words, the goal is to identify "guilty" model elements that cause trouble. However, it is equally important in the debugging process to identify the "innocent" model elements that definitively do not contribute to the fault. For instance, when using general-purpose programming languages, it is common to comment portions of code to see whether the failure is still present

without these portions. In a more advanced fashion, unit testing aims at testing independently the different parts of a system—for instance using partial and trusted implementations called *stubs*—in order to mark as many units as possible "innocent", and to eventually isolate the guilty unit responsible for a failure.

VI. Conclusion and Acknowledgements

While verification and validation is a necessary step to identify the defects in models, *debugging* is a crucial activity when it comes to dealing with such defects to improve the quality of models. The International Workshop on Debugging in Model-Driven Engineering (MDEbug) aims at providing an event for the community to share ideas and results in this research area. This first edition was well attended throughout the day, and the afternoon discussions were both lively and constructive, leading to a wide range of potential research topics. In addition, we observed that debugging was a quite recurrent topic throughout the presentations of the MODELS 2017 conference, which strengthens our belief that it is an important research direction for the success of MDE.

We thank everyone who took part in the success of the workshop, including the program committee members, the authors, our keynote speaker, and everyone who attended the workshop or took part in the discussions.

References

[1] S. Gabmeyer, P. Kaufmann, M. Seidl, M. Gogolla, and G. Kappel, "A feature-based classification of formal verification techniques for software models," *Software & Systems Modeling*, Mar 2017. [Online]. Available: https://doi.org/10.1007/s10270-017-0591-z

[2] F. Hilken, M. Gogolla, L. Burgueño, and A. Vallecillo, "Testing models and model transformations using classifying terms," *Software & Systems Modeling*, Dec 2016. [Online]. Available: https://doi.org/10.1007/s10270-016-0568-3

[3] K. Zurowska and J. Dingel, "Language-specific model checking of UML-RT models," *Software & Systems Modeling*, vol. 16, no. 2, pp. 393–415, May 2017. [Online]. Available: https://doi.org/10.1007/s10270-015-0484-y

[4] E. Planas, J. Cabot, and C. Gómez, "Lightweight and static verification of UML executable models," *Computer Languages, Systems & Structures*, vol. 46, no. Supplement C, pp. 66 – 90, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1477842415300361

[5] Peggy Aldrich Kidwell, "Stalking the Elusive Computer Bug," *IEEE Annals Of The History Of Computing*, vol. 20, no. 4, pp. 3–7, 1998.

[6] A. Zeller, *Why Program Fail – 1st Edition*. Elsevier, 2004. [Online]. Available: http://www.whyprogramsfail.com/

[7] N. Bandener, C. Soltenborn, and G. Engels, "Extending DMM Behavior Specifications for Visual Execution and Debugging," in *Proceedings of the Third International Conference on Software Language Engineering (SLE'10)*, vol. 6563 LNCS. Springer Berlin Heidelberg, 2010, pp. 357–376.

[8] T. Mayerhofer, P. Langer, and G. Kappel, "A runtime model for fUML," in *Workshop on Models@run.time (MRT)*. ACM, 2012, pp. 53–58.

[9] S. Van Mierlo, Y. Van Tendeloo, and H. Vangheluwe, "Debugging Parallel DEVS," *SIMULATION*, vol. 93, no. 4, pp. 285–306, 2017. [Online]. Available: http://dx.doi.org/10.1177/0037549716658360

[10] J. Schönböck, G. Kappel, M. Wimmer, A. Kusel, W. Retschitzegger, and W. Schwinger, "Tetrabox - a generic white-box testing framework for model transformations," in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1, Dec 2013, pp. 75–82.

[11] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux, "Kompren: Modeling and Generating Model Slicers," *Software and Systems Modeling*, Oct. 2012. [Online]. Available: http://hal.inria.fr/hal-00746566/PDF/slicer.pdf

[12] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting Efficient and Advanced Omniscient Debugging for xDSMLs," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2015. New York, NY, USA: ACM, 2015, pp. 137–148.

[13] J. Corley, B. P. Eddy, E. Syriani, and J. Gray, "Efficient and scalable omniscient debugging for model transformations," *Software Quality Journal*, pp. 1–42, 2016.

[14] R. van Rozen and T. van der Storm, "Toward live domain-specific languages," *Software & Systems Modeling*, Aug 2017. [Online]. Available: https://doi.org/10.1007/s10270-017-0608-7

[15] B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux, "Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification," *Journal of Software*, vol. 4, no. 9, pp. 943–958, 2009.

[16] H. Wu, J. Gray, and M. Mernik, "Grammar-driven generation of domain-specific language debuggers," *Software: Practice and Experience*, vol. 38, no. 10, pp. 1073–1103, 2008.

[17] S. Mustafiz and H. Vangheluwe, "Explicit modelling of statechart simulation environments," in *Proceedings of the 2013 Summer Computer Simulation Conference*, ser. SCSC '13. Vista, CA: Society for Modeling & Simulation International, 2013, pp. 21:1–21:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2557696.2557720

[18] Y. Laurent, R. Bendraou, and M. Gervais, "Executing and debugging UML models: an fUML extension," in *Symposium on Applied Computing (SAC)*. ACM, 2013, pp. 1095–1102.

[19] J. Denil, H. Kashif, P. Arafa, H. Vangheluwe, and S. Fischmeister, "Instrumentation and preservation of extra-functional properties of simulink models," in *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, ser. DEVS '15. San Diego, CA, USA: Society for Computer Simulation International, 2015, pp. 47–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=2872965.2872972

[20] A. Krasnogolowy, S. Hildebrandt, and S. Wlitzoldt, "Flexible Debugging of Behavior Models," in *International Conference on Industrial Technology (ICIT)*. IEEE, 2012, pp. 331–336.

[21] A. Chiş, M. Denker, T. Gîrba, and O. Nierstrasz, "Practical domain-specific debuggers using the moldable debugger framework," *Comput. Lang. Syst. Struct.*, vol. 44, no. PA, pp. 89–113, Dec. 2015.

[22] R. T. Lindeman, L. C. L. Kats, and E. Visser, "Declaratively Defining Domain-Specific Language Debuggers," in *International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2012, pp. 127–136.

[23] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Melange: A meta-language for modular and reusable development of DSLs," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2015. New York, NY, USA: ACM, 2015, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2814251.2814252

[24] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini, "Reusable components of semantic specifications," in *Transactions on Aspect-Oriented Software Development XII*, S. Chiba, É. Tanter, E. Ernst, and R. Hirschfeld, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 132–179. [Online]. Available: https://doi.org/10.1007/978-3-662-46734-3_4

[25] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution Framework of the GEMOC Studio (Tool Demo)," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2016. New York, NY, USA: ACM, 2016, pp. 84–89. [Online]. Available: http://doi.acm.org/10.1145/2997364.2997384